# ASSIGNMENT 1: REINFORCEMENT LEARNING

**Sherry Usman**
4168216

## 1 DYNAMIC PROGRAMMING

1. To implement the Q iteration algorithm with the help of dynamic programming it is necessary to update the select action function in order to implement the greedy policy (choosing an action where the reward is highest). The equation for greedy action selection can be seen below.

$$\pi(a|s) = max_a Q(s,a) \tag{1}$$

where $s$ is the state, $a$ is the action and $\pi(a|s)$ is the probability of taking the action $a$ with the state $s$ under stochastic policy.
For this I used the helper function called full_argmax() defined in Helper.py to find all points (or actions) in the array Q(s,a) where the resulting reward Q(s,a) is the maximum. After implementing the select action function I revised the update() function to implement the Q-iteration update as shown in the algorithm below.

**Algorithm 1:** Tabular Q-value iteration (Dynamic Programming)

**Input:** Threshold $\eta \in R^+$.
**Result:** The optimal value function $Q^\star(s,a)$ and/or associated optimal policy $\pi^\star(s)$.
**Initialization**: A state-action value table $\hat{Q}(s,a) = 0 \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$
**repeat**
  $\Delta \leftarrow 0$
  **for** *each* $s \in \mathcal{S}$ **do**
    **for** *each* $a \in \mathcal{A}$ **do**
      $x \leftarrow Q(s,a)$ /* Store current estimate */
      $\hat{Q}(s,a) \leftarrow \sum_{s'} \left[ p(s'|s,a) \cdot \left( r(s,a,s') + \gamma \cdot \max_{a'} Q(s',a') \right) \right]$ /* Eq. 1 */
      $\Delta \leftarrow \max(\Delta, |x - \hat{Q}(s,a)|)$ /* Update max error */
    **end**
  **end**
**until** $\Delta < \eta$;
$Q^\star(s,a) = \hat{Q}(s,a)$ /* Converged at optimal value function */
$\pi^\star(s) = \arg\max_a Q^\star(s,a) \quad \forall s \in \mathcal{S}$ /* Optimal policy is greedy */
**Return** $Q^\star(s,a)$ and/or $\pi^\star(s)$.

Figure 1: Tabular Q learning algorithm for Dynamic Programming (Plaat, 2023)

As seen above this algorithm uses two nested for-loops to sweep through each individual state-action pair, updating Q(s,a) and the threshold $\Delta$ as it goes along. The updated value of the Q(s,a) is the summation for all subsequent states s', the probability of transitioning from state s to s' with action $a$ denoted by $p(s'|s,a)$, multiplied by the reward received by transitioning from state $s$ to $s'$ with action $a$ denoted by $r(s,a,s')$ subtracted by the discount factor $\gamma \in [0,1]$ which controls how much the agent cares about the immediate reward compared to long term rewards.

2. As seen below the Q values at the start of the nested for loop have a large variance across state-action pairs, ranging from -3 to 96.7. However, as the number of iterations increases the Q values become to converge to a point with lower variance as shown in figure 3, ranging from 82.7 to 99.8.
The final Q values for each state action pair show the rewards expected for choosing that particular action in that particular state. Thus, with that in mind and working backwards from the goal state we can see that the expected rewards are higher closer to the goal state (highest is 99.8) and get progressively lesser as you move away from the goal state in both horizontal and vertical axes. This is because we assign equal credit to each action taken

to get to the final goal state even though there might have been one action that was more critical than the other. Thus, the expected return then becomes the total return divided by the number of steps. Therefore as you go further back the number of steps increase and the expected return decreases.

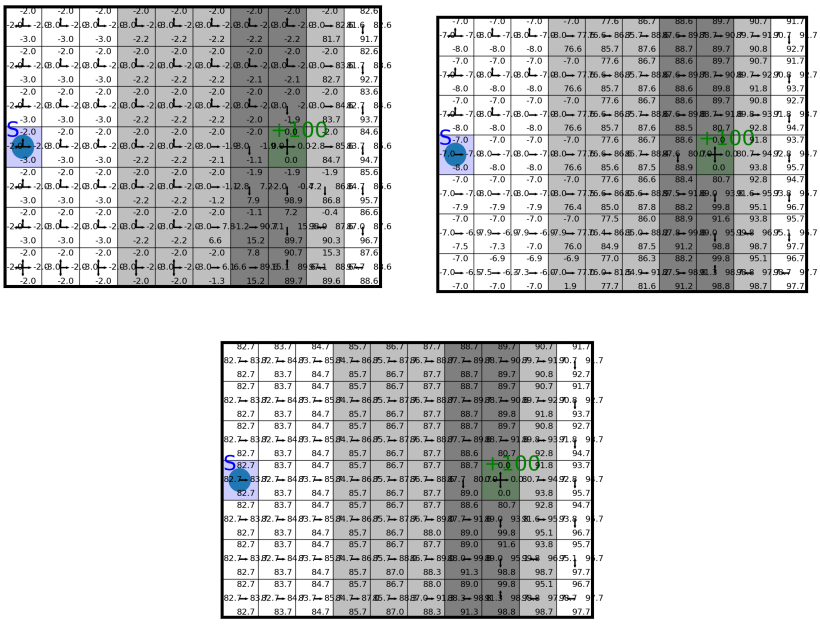

Figure 2: Q value table at the start, middle and end of the nested for loop.

3. The value of V(s,a) at the point s = 0 or location (0,3) is the maximum Q(s,a) for a particular state-action taken from the start state. This is 83.8 or the action to the right as shown in Figure 3. This makes logical sense as a step towards the right from the start state brings the agent closer to the goal state (7,3), compared to the other actions where the resulting state does not bring the agent physically closer to the goal.

4. To calculate the average reward per step it is important to first calculate the the average number of steps over a number of repetitions for e.g. 100. The average number of steps denoted by $\bar{x}$ is 17.42. The average total reward denoted by $r_{\bar{100}}$ with 100 repetitions is 1600.91. Thus the average reward per time step is

$$\frac{r_{\bar{100}}}{\bar{x}} = \frac{1600.91}{17.42} = 91.86 \tag{2}$$

However, we have to subtract this by the converged optimal value at the start to give us the reward per step (91.86 - 83.8 = 8.06).

```
# if s is goal state (terminal) make it a self-loop without rewards
state_is_a_goal = np.any([np.all(goal_location == s_location) for goal_location in self.goal_locations])
if state_is_a_goal:
    # Make actions from this state a self-loop with 0 reward.
    p_sas[s,a,s] = 1.0
    r_sas[s,a,] = np.zeros(self.n_states)
```

Figure 3: Q value table at the convergence/end of the nested for loop.

5. The figure above is a code snippet above from Environment.py and shows how this implementation deals with the fact that the goal state is a terminal state (causes the program to stop running). It does this by checking if the current state is a goal state and if so setting the probability of going from that state to itself as 1, meaning that after the agent reaches that state it does not go anywhere. Furthermore the reward for all actions at the goal state is turned to 0. The agent still converges because this is only implemented when the agent reaches the goal state and not before that. Another way of making the goal state a terminal state is by setting the reward of taking an action a after reaching a goal state to negative infinity.
6. The key noticeable difference in changing the goal state from (7,3) to (6,2) is that the agent is now finding a more efficient approach towards the goal state and not going all the way around it as in the previous case. This means that the exploration has decreased and the agent is exploitin the learning from the previous episode.

## 2 EXPLORATION

1. In the second question we switch to a model-free reinforcement learning setting. This means we no longer have access to the StochasticWindyGridWorld environment and can no longer sweep through all states to compute Q(s,a) for individual state-action pairs. Thus we need to introduce some exploration to find the greedy policy. The ration of exploration and exploitation defined in the action selection function can be implemented in two different ways by two different policies: E-Greedy and Boltzmann.

   We first implement the e-greedy policy. The E-Greedy policy uses an epsilon parameter $\epsilon$ to determine the ratio of exploration to exploitation. An $\epsilon$ value of 0 gives a greedy policy identical to equation (1) while an epsilon value of 1 gives a uniform/random policy. As shown in the algorithm below we take a small random value n between 0 and 1. If n is less than $\epsilon$ we take a random action from the state space and otherwise we take a greedy action or an action which returns the highest Q(s,a) for the individual state-action pair.

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon \cdot \frac{|A|-1}{|A|} & if a = argmax_{b \in A} \hat{Q}(s,b) \\ \frac{\epsilon}{|A|} & otherwise \end{cases} \tag{3}$$

   The Boltzmann policy uses a temperature parameter $\tau$ and this temperature parameter allows you to scale the exploration versus exploitation. As $\tau$ tends to $\infty$ the policy becomes more uniform and random as the agent increasingly chooses the explorative action over the exploitative action. As $\tau$ tends to 0, the agent increasingly chooses the exploitative greedy action. This relationship is shown in the equation below.

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{b \in A} e^{Q(s,b)/\tau}} \tag{4}$$

   The second part of this algorithm is the update method, namely how to transfer learning of the new child state to the parent state. In this question we will implement the 1-step Q learning explained below.
   The one-step q learning has two significant parts. We first compute **the backup estimate target** $G_t$, the formula for which is shown below.

$$G_t = r_t + \gamma * \max_{a'} \hat{Q}(s_{t+1}, a') \tag{5}$$

   $G_t$ is the sum of reward $r_t$ and the learning rate $\gamma$ multiplied with the maximum $\hat{Q}$ value for the next state $s_{t+1}$ and the next action $a'$. Then we compute **the tabular learning update** as shown below.

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha * [G_t - \hat{Q}(s_t, a_t)] \tag{6}$$

**Algorithm 2:** Tabular Q-learning.

**Input:** Exploration parameter, learning rate $\alpha \in (0, 1]$, discount parameter $\gamma \in [0, 1]$, total *budget*.

$\hat{Q}(s, a) \leftarrow 0, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}.$ /* Initialize Q-value table */

$s \sim p_0(s)$ /* Sample initial state */

**while** *budget* **do**

  $a \sim \pi(a|s)$ /* Sample action, e.g., $\epsilon$-greedy, softmax */

  $r, s' \sim p(r, s'|s, a)$ /* Simulate environment */

  $\hat{Q}(s, a) \leftarrow \hat{Q}(s, a) + \alpha \cdot [r + \gamma \cdot \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a)]$ /* Q update */

  **if** $s'$ *is terminal* **then**

    $s \sim p_0(s)$ /* Reset environment */

  **else**

    $s \leftarrow s'$

  **end**

**end**

**Return**: $\hat{Q}(s, a)$

Figure 4: Q value table at the middle of the nested for loop.

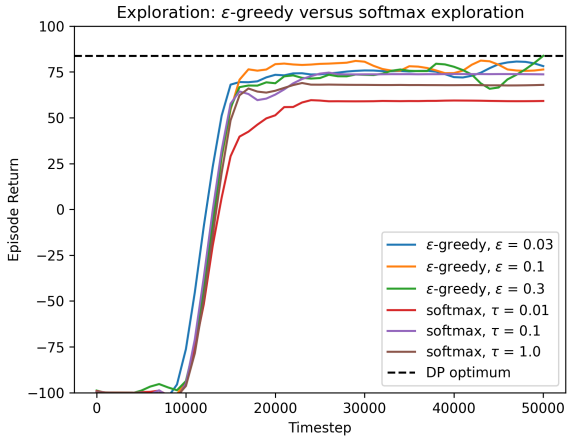


Figure 5: Comparison of E-greedy policy episodic returns for each value of $\epsilon$ and $\tau$

2. Figure 5 compares the averaged episodic returns for both the E-Greedy and the Boltzmann softmax policy with varying values of $\epsilon$ and $\tau$ over 50000 timesteps averaged over 20 repetitions. From the figure we can see that for both policies, as the number of timesteps increases the average episodic return also increases showing that the agent gets better at choosing the trace with the highest possible returns. Furthermore, we see that when choosing $\epsilon$ and $\tau$ there seems to be a sweet spot with middle values 0.1 giving highest returns then lower and higher values. Furthermore, we see that when comparing the e-greedy policy with the softmax policy as a whole, the e-greedy policy performs better on average than the softmax policy. When comparing this with the optimal value of DP we see that that is slightly higher at 83.8. However, this does not mean that the Dynamic Programming algorithm is better on a whole since it is model-based and assumes that the agent knows every reward for every state-action pair which is not representative of real-life decision problems.

3. Adding one more goal state makes the episodic returns for varying values of epsilon and tau much smaller. This is because the agent is now split between two different goal states and explores significantly more takes previously unexplored paths. However this exploration is not always fruitful as it does not always lead the agent to finding the goal state/states.

## 3 On and Off-Policy

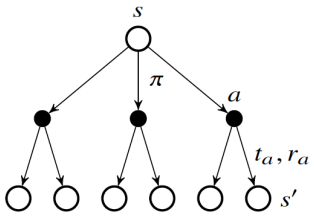


Figure 6: MDP transition diagram (Plaat 2023)

1. Figure 6 shows that as the agent travels downwards, the agent's action selection moves downwards and with each action the agent learns the reward values of the new child state and propagates that information called learning back to the previous parent state. There are two main methods that can be used to propagate information backwards, on policy methods such as SARSA and off-policy methods such as Q-learning.
   On-policy methods like Q-learning only propagates the reward of the best possible action to the previous parent state and therefore attempts to learn the optimal policy. Their formula is discussed in (5) and (6). Off-policy methods such as SARSA on the other hand propagate the reward of the current state to the previous parent state, irrespective of the whether the current state is optimal or not. Therefore SARSA attempts to learn the value function of the policy we actually execute. Thus we can define the backup equation for SARSA as

$$G_t = r_t + \gamma * \hat{Q}(s_{t+1}, a_{t+1}) \tag{7}$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \left[ G_t - \hat{Q}(s_t, a_t) \right] \tag{8}$$

   The algorithm for tabular SARSA GOES HERE

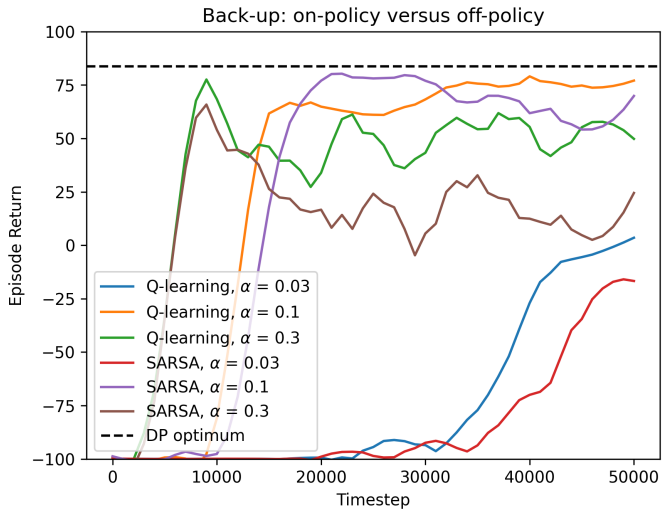


Figure 7: Comparison of on and off policy methods for varying values of learning rate $\alpha$

2. In figure 7 we analyse the overall performance of Q-learning and SARSA over a period of time, for example a sequence of timesteps at 50,001 and with varying learning rates denoted by $\alpha$. We observe that the episodic returns for Q-learning and SARSA are higher at higher levels of $\alpha$ and vice-versa.

However, optimal returns are achieved when the learning rate is not too low and not too high, at $\alpha$ = 0.1 as at higher values (0.3) we get diminishing episodic returns. We can also see here that the episodic returns gained from the initial Dynamic Programming algorithm depicted by the dotted black line exceeds the Q-learning and SARSA agorithms.

## 4 Depth of Target

In the previous implementations we were bootstrapping our learnings after every step or action. However, it is also possible to bootstrap our learnings after more than one step, taking the sum of the rewards achieved in this case. This number of steps $n$ is called **depth of target**, and it leads to the n-step Q learning method. Using n step Q learning we compute the target $G_t$ as

$$G_t = \sum_{i=0}^{n-1} (\gamma)^i * r_{t+1} + \gamma^n * \underset{a}{max} Q(s_t + n, a) \tag{9}$$

Then we add our target $G_t$ values to our Q(s,a) values using the formula below.

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \left[ G_t - \hat{Q}(s_t, a_t) \right] \tag{10}$$

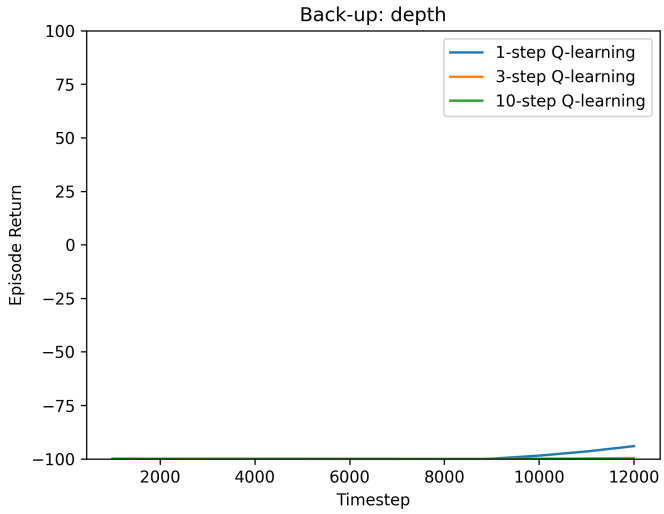


Figure 8: MDP transition diagram (Plaat 2023)

Unfortunately, I was obtain a perfect graph for Depth of Target as my update function kept throwing out null values for Q(s,a). I see this as evidence that even in low-dimensional environments, n-step and montecarlo techniques quickly become inefficient because of their need for extra memory when it comes to storing past states, rewards and actions. Furthermore, 1-step Q learning tends to outperform other algorithms.

## 5 Reflection

1. **A**s we can see in the previous figures, the dynamic programming algorithms tend to outperform the reinforcement learning algorithms. However, it should be noted that in dynamic programming the agent has a model of the environment and therefore knows all the variables needed to make the right action such as the reward/cost for every state-action pair. However, this is not representative of real-life sequential decision problems and does not reflect real learning on the part of the agent.

2. Figure 7 shows that e-greedy policies are more stable and converge to an optimal value faster than softmax policies. Furthermore since that difference is small and can be mitigated by choosing the right temperature parameter $\tau$, I would say I prefer Boltzmann policy over e-greedy policy. While Boltzmann is more complex, its parameter temperature is more flexible than epsilon and allows for smoother trade-off between exploration and exploitation.
3. 
4. John might find that it is a discount factor of 1.0 as more beneficial in this environment as it means that the agent considers future and immediate rewards as equally beneficial. This is true for the case where there is no penalisation for a step as this encourages exploration and traversing newer traces in the Markov Decision Process.
5. On policy methods like SARSA update their learnings based on the behavior policy they are following and the actions they take. Therefore, they tend to be more stable than Q learning which are looking for the optimal policy making the target keep changing. However, SARSA often can have the problem of getting stuck at a local optima as it looks for optimal returns in the policy it is implementing. Furthermore, since SARSA only learns from the policy it is following it does not explore state-action pair and therefore may need to run for longer time periods to accurately estimate the rewards from each state-action pair. Furthermore the learning from SARSA might be more limited and therefore they cannot reuse their past learnings as well as off-policy methods like Q-learning.
6. One of the more obvious benefits of n step Q-learning and Monte-Carlo is that we no longer need to update the Q value table after every step and we rely more on the previous experiences of the agent to assign credit to state-action pairs. Furthermore, since we now compute rewards over a number of steps we have lesser variance making it better than 1-step Q learning in more stochastic environments with higher noise. 1-step Q-learning methods have the benefit of having low bias as we update the Q value table after every step. However n-step Q learning methods have higher bias as every n steps taken are assigned the same credit irregardless of whether each were equally crucial in getting to the goal state.
   Monte-Carlo algorithms are another extreme and do not perform very well as they update the Q value table only after the end of an episode and assign equal credit/reward to all the previous steps taken irregardless of whether that step was crucial in getting to the goal state. Thus they have an extremely high bias but a low variance. We can say that Monte Carlo and 1-step Q learning are two extremes: the first has extremely low variance but really high bias, the second with low bias but high variance.
7. Tabular q learning algorithms are good for high accuracy as they can give estimates of Q(s,a) for each state-action pair by sweeping through each state-action pair. Tabular q learning algorithms can be feasible if your environment is low-dimensional (has a few number of states and a few actions for each state). An example for a low-dimensional environment is a 3 by 3 tic-tac-toe environment where there are 9 states and 3 actions per state, meaning it is $3^9$ state-action pairs in total. However, once we enter higher-dimensional problems such as a robotic arm or a chess board the number of states and actions increases, the complexity of the tabular algorithm increases and much more computational memory is required to store the Q value table. Thus tabular q learning algorithms might be accurate in low dimensional environments but are not always scalable.

## REFERENCES

Aske Plaat. *Deep Reinforcement Learning*. 2023.